

Re-configurable Distributed Scripting

M.Ranganathan, Laurent Andrey, Virginie Schaal and Jean-Philippe Favreau
{mranga andrey schaal favreau}@antd.nist.gov

National Institute of Standards and Technology

820 W. Diamond Ave
Gaithersburg, MD 20899, U.S.A.

Several distributed testing, control and collaborative applications are reactive or event driven in nature. Such applications can be structured as a set of handlers that react to events and that in turn can trigger other events. We have developed an application building toolkit that facilitates development of such applications. Our system is based on the concept of Mobile Streams. Applications developed in our system are dynamically extensible and re-configurable and our system provides the application designer a mechanism to control extension and re-configuration. We describe our system model and give examples of its use.

Introduction

Consider the problem of testing a distributed database system. For such a system, a reasonable test sequence may consist of the following actions - (1) Insert a record into the database (2) Concurrently modify the database record from multiple locations (3) Check the consistency of the database at the end of the operation. We may wish to automate this test by scripting it from some central location - for example storing the test on a server. This raises several complications: (1) The physical locations (IP addresses) of the machines participating in the test may not be known a-priori. We need to be able to structure the test using logical locations that get mapped to physical ones at a later time (2) The participating machines may abort the test at any time. (3) The test must have some integrity guarantees to ensure its validity. In such a system we can think of synchronization actions, failures, machines joining the test and so on as "events" that drive the test.

Next, consider a collaborative application such as a network chat. For such a system, participants may join and leave or turn off their machines at any time. The physical locations and machine resources of the participants is not known a-priori but the distributed application "structure" is known. User actions such as typing on the keyboard, joining and leaving the chat and so on generate "events" to which the distributed system is expected to respond.

A large class of collaborative, testing, monitoring and control applications fit the "event-driven" paradigm. An event-driven application is driven by asynchronous external inputs that cause event-handlers to be invoked. In a collaborative application, events may be triggered by user actions. In a distributed monitoring and control system, events may be triggered by transducer inputs. In a distributed testing scenario, events are triggered by test outputs, timer timeouts and so on. In this paper, we describe *AGNI*¹ - a Tcl-based middle-ware for scripting distributed, dynamically re-configurable, event-oriented applications. Our system incorporates the following (1) Separation of logical application structure from physical implementation (2) Dynamic extensibility (3) Dynamic Re-configurability and (4) Failure detection.

AGNI is based on the *Mobile Stream* paradigm. A Mobile Stream (*MStream*) is a named communication end-point in a distributed system that can be moved from machine to machine as computation is in progress while maintaining a specified ordering guarantee. An *MStream* has a globally unique name and may be located on any machine that runs an execution environment for it and allows it to be moved there. Multiple event-handlers (*Handlers*) may be dynamically attached to (and detached from) an *MStream* and are independently and concurrently invoked for each message. Handlers operate in an atomic fashion. By

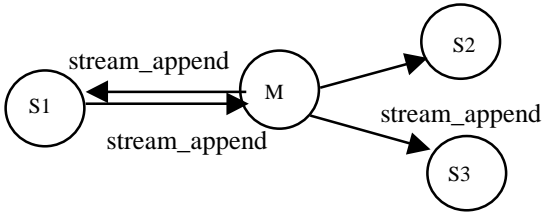
¹ Agents at NIST. Also Sanskrit for "fire".

“atomic” we mean that changes in the state of the MStream (which includes various attributes such as its location, the set of Handlers attached to it and so on) are deferred until the time when the Handlers complete execution. Handlers are grouped and are registered for specific events by *Agents*. An Agent initializes the execution environment for the event handlers that it registers. Each MStream may have several Agents associated with it. Streams are opened (analogously to files) with a specified ordering guarantee that is preserved despite dynamic re-configuration. When a MStream moves, it carries its Agents and any global state that the Agents declare to be of interest to the new location and re-initializes these at the target location of the move.

The closest analogy to an MStream is a "mobile, active mailbox". As in a mailbox, an MStream has a globally unique name. MStreams provide a *FIFO* ordering guarantee – messages are consumed in the same order in which they were sent. Message delivery at an MStream results in a message handler being invoked - which makes them analogous to an "active mailbox". Usually mailboxes are stationary. MStreams, on the other hand, can migrate from host to host - which makes an MStream analogous to a "mobile, active mailbox". We clarify these notions with an example.

A Self-reconfiguring Chat

Consider a chat application where participant in the chat can send messages to all other participants. In such an application, it may be desired that all participants see an evolving conversation in the same order – necessitating that messages be consumed by each participant in the same order.

	<pre> stream_open M set consumers [list S1 S2 S3] register_agent M \$consumers { global counter global consumers set consumers \$argv foreach consumer \$consumers { set counter(\$consumer) 0 ;# 1.b.1 stream_open \$consumer ;# 1.b.2 } set count 0 on_stream_append { ;# 1.b.3 incr count set loc [stream_sender_name] incr counter(\$loc) foreach stream \$consumers { stream_append \$stream \$argv ;# 1.b.4 } if { \$count == 50 } { #find_max finds the max counter value set newloc [find_max counter] stream_relocate [stream_location \$newloc] ;#1.b.5 } } on_stream_relocation { ;# 1.b.6 foreach consumer \$consumers { set counter(\$consumer) 0 ;# 1.b.7 } set count 0 } } </pre>
<p>Figure 1.a: Structure of chat application</p>	<p>Figure 1.b: Chat application mobile dispatcher.</p>
<pre> set my_id [my_id] ;# 1.c.1 set mydisplay [stream_open S\$my_id] ;# 1.c.2 register_agent \$mydisplay "" { ;# 1.c.3 on_stream_append { puts stdout \$argv ;# 1.c.4 } } </pre>	
<p>Figure 1.c: Chat application display handler.</p>	
<pre> set dispatcher [stream_open M] while {1} { set input [gets] ;# 1.d.1 stream_append \$dispatcher \$input ;# 1.d.2 } </pre>	
<p>Figure 1.d: Chat application input handler.</p>	

Global ordering in this application is ensured by each participant sending messages to a central mobile MStream M that then re-broadcasts it to each participant. The scheme is shown in figure 1.a. We assume that there are three participants in the conversation labeled 1,2 and 3. Each participant i hosts an MStream S_i on her workstation. This purpose of this MStream is to display output. There is a central MStream M has a handler that simply re-sends each message sent to it to each of the “display” MStreams (S_i). Each participant starts an input handling loop (Figure 1.d) that reads input from the keyboard and sends the input to M . The Agent for M (Figure 1.b) specifies initialization code that initializes a counter array (#1.b.1), opens each of the display streams (#1.b.2) and registers *on_stream_append* and *on_stream_relocation* handlers(#1.b.3 and #1.b.6). When a message is delivered to the MStream, M , its *stream_append* handler is run. It responds by re-dispatching the message to each of the display MStreams (#1.b.4). Every 50 messages it moves M to the location from which the most messages have originated (#1.b.5). Repositioning M in this fashion, reduces the round-trip latency for the most active user. When M arrives at the new location its *on_stream_relocation* handler is run. The *on_stream_relocation* handler just zeros the message counters at the location where it arrives (#1.b.7). The remaining pieces of the application consist of the display handlers and the input handlers. The input handler (Figure 1.d) reads input from the keyboard (#1.d.1) and appends it to M (#1.d.2). The display handler (Figure 1.c), first inquires its own id (#1.c.1) and opens a display handler MStream (#1.c.2). It then attaches a display handler Agent to the MStream (#1.c.3) that specifies an *on_stream_append* handler. The registered handler simply prints any data sent to it (#1.c.4).

We have developed a toolkit to share the GUI of unmodified Tk applications based on this simple application structure.

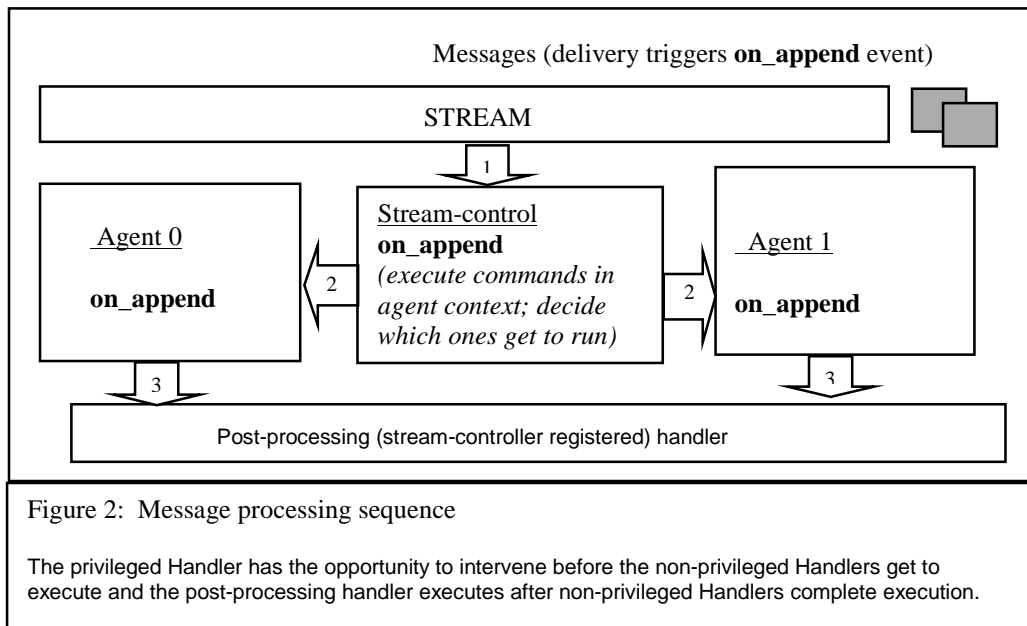
Configuration Control

Dynamic extension and re-configuration is useful in scripting distributed applications. However, to ensure system integrity, we need to be able to place controls on how the system can be extended and re-configured. In AGNI, this is done at three “levels” - at the “System-wide” level, at the “Site-level” and at the level of individual Streams.

Each workstation that wishes to participate in the system runs an *Agent Daemon* that has a unique location identifier assigned to it by a reliable and secure *Session Leader Agent Daemon*. The configuration control policy in AGNI is implemented by the Session Leader, the Agent Daemons and by the MStreams themselves. In keeping with the event-driven design of the system, the control mechanism is also event-driven. There are three types of control events : (1) *Stream Control* events that allow control over stream-specific actions such as Agent attachments and Opens; (2) *Site-specific* events that allow controls to be placed over admitting an MStream to a site and access to resources (such as files) at the site and (3) *System-wide Control* events that allow controls to be placed over MStream additions and deletions, new Agent Daemon additions and so on.

Each Agent Daemon may be started with an *Agent Daemon Script* that implements a site specific policy and the Session Leader may also specify a *System Control Script* that implements the system-control policy. Each MStream may have a “privileged” Agent called a *Stream Controller* that may register “privileged handlers”. All other handlers are “non-privileged”. The idea is to allow the privileged handler to execute before non-privileged handler when an event occurs - thereby giving the privileged handler the ability to control the environment in which the non-privileged handler executes.

For example, Figure 2 shows what happens during the common case of data consumption at an MStream. First, the privileged *on_append* handler gets control if it is present. This Handler may select the non-privileged *on_append* handlers of the MStream to which the message is then delivered. Before doing this, it may modify the execution environment of the non-privileged handlers. The Stream Controller may also register a *post-processing* handler. After the non-privileged handlers have completed execution, the post-processing handler gets control and has the opportunity of limiting side effects such as stream re-location actions specified by the non-privileged handlers. The Stream Controller can register handlers that intervene in various other events such as MStream motion, new Agent attachments, stream destruction and so on.



Related Work, Future Work and Acknowledgements

The design presented in this paper has been influenced by several other systems such as *TACOMA*[1], *Voyager*[2] and *Aglets*[3] as well as the first author's experience with the *Sumatra* system[4]. In contrast with these works, we focus on distributed scripting - our view being that mobility is a useful tool for configuration of distributed systems rather than a way to replace such systems. We also differ significantly in the configuration control mechanisms that we have described above.

We have used our system to script various distributed applications including the distributed record and replay of the MITRE XCVW collaborative tool and distributed testing of the MASH toolkit. Our planned focus for future work is : (1) use of multicast to improve scalability; (2) a more robust security model (3) application development with AGNI.

This work was supported by DARPA funding under the IC & V program. Several aspects of our design are an evolution of ideas and experience gained from previous work with Anurag Acharya of UCSB [4]. We are indebted to Charles Crowley for making the source code of *Tk Replay* [5] freely available and to Scriptics Corp. [6] for Tcl/Tk.

Our system runs on Solaris and LINUX and we are currently porting it to Windows NT. A Beta version of AGNI may be downloaded from <http://snad.ncsl.nist.gov/antd-staff/mranga/AGNI/AGNI.html>

References

- [1] D.Johansen, R. van-Renesse, F.B.Schnieder, "An introduction to the TACOMA Distributed System", *Technical Report 95-23*, Dept of Computer Science, University of Tromso, Norway, 1995.
- [2] <http://www.objectspace.com/products/voyager/index.html>
- [3] B. Venners, "Under the Hood: The architecture of Aglets", *Java World*, April 1997.
- [4] M. Ranganathan A.Acharya S.D.Sharma and Joel Saltz, "Network-aware Mobile Programs", *USENIX 97*.
- [5] C. Crowley, "Tk-Replay: Record and Replay in Tk", *USENIX Third Annual Tcl/Tk Workshop*, 1995.
- [6] <http://www.scriptics.com>